



LIVRO

UNIDADE 2

Sistemas Operacionais

Sistemas Operacionais

Cynthia da Silva Barbosa

© 2018 por Editora e Distribuidora Educacional S.A.

Todos os direitos reservados. Nenhuma parte desta publicação poderá ser reproduzida ou transmitida de qualquer modo ou por qualquer outro meio, eletrônico ou mecânico, incluindo fotocópia, gravação ou qualquer outro tipo de sistema de armazenamento e transmissão de informação, sem prévia autorização, por escrito, da Editora e Distribuidora Educacional S.A.

2018

Editora e Distribuidora Educacional S.A.
Avenida Paris, 675 – Parque Residencial João Piza
CEP: 86041-100 – Londrina – PR
e-mail: editora.educacional@kroton.com.br
Homepage: <http://www.kroton.com.br/>

Sumário

Unidade 2 Processos e Threads 2	5
Seção 2.1 - Processos	7
Seção 2.2 - Comunicação entre processos	21
Seção 2.3 - Escalonamento de processos	38

Processos e Threads 2

Convite ao estudo

Caro aluno, seja bem-vindo!

Nesta unidade veremos: o conceito, as características e o funcionamento de processo e threads e como eles são implementados, como os processos são criados e finalizados pelo sistema operacional e a hierarquia e os estados de processos. Um processo pode ser definido como um programa em execução. Porém, o sistema operacional não executa apenas os programas que solicitamos que sejam rodados.

Por exemplo, quando abrimos uma página em um navegador, ela pode acionar outros processos para que seja devidamente carregada, sem impactar as demais abas abertas. Além disso, existem outros processos executados no computador que não são propriedade do usuário, mas que são acionados para garantir a performance do sistema operacional, como o antivírus e o gerenciamento da rede.

Após o término desta unidade, você terá condições de entender o funcionamento e a implementação dos processos e threads. Aprendendo sobre os processos, você terá um conhecimento sobre o Gerenciador de Tarefas de seu sistema operacional, o que o ajudará a manter o bom funcionamento do seu computador.

Neste contexto, Lucas acabou de ser contratado como gerente da área de Tecnologia da Informação de uma empresa prestadora de serviços hospitalares. Além de fornecer suporte aos usuários dos sistemas hospitalares, outra função realizada na empresa é fazer pesquisas na internet sobre a qualidade de serviços na área da saúde, baixar livros e arquivos da área que auxiliam o desenvolvimento de aplicativos em TI e editar textos sobre o processo de avaliação e certificação da qualidade dos serviços hospitalares. Sempre que uma dessas atividades realizadas por Lucas é executada, um processo é criado e

ele pode chamar outros processos para auxiliá-lo por meio da transferência de informações entre eles. Para escolher qual processo será executado, o escalonador é acionado pelo sistema operacional e define os critérios de acesso à CPU que cada processo terá para que não afete a execução das aplicações. Ao receber um chamado de atendimento de suporte aos usuários dos sistemas hospitalares, um estagiário da equipe liderada por Lucas o aciona para ir até a sala do usuário que realizou a abertura do chamado. Durante o percurso, Lucas é questionado sobre como os processos são criados e finalizados, sobre como o sistema operacional trata a hierarquia de processos e seus estados, sobre o funcionamento da comunicação entre processos e seus problemas clássicos da comunicação e sobre como trabalham o escalonador de processos e os algoritmos de escalonamento. Para entender melhor o funcionamento dos processos e os algoritmos de escalonamento, Lucas sugere ao estagiário que seja implementado um algoritmo de escalonamento de processos por meio de semáforos.

Vamos juntos conhecer mais sobre essas questões através de algoritmos!

Seção 2.1

Processos

Diálogo aberto

Caro aluno, os processos são programas ou tarefas em execução e o sistema operacional é o responsável por administrá-los, por meio do gerenciador de processos. Existem os processos iniciados pelo usuário, como executar um editor de textos, abrir uma página na internet, abrir o aplicativo de músicas, entre outros. Há também os processos iniciados por outros processos, por exemplo, uma página da internet solicitando a ajuda de outro processo para fazer o carregamento dos seus elementos. Nessa seção você conhecerá o conceito, as características, a hierarquia e os estados dos processos e threads e como se dá a criação e o término de processos. Relembrando nosso contexto, Lucas acabou de ser contratado como gerente da área de Tecnologia da Informação de uma empresa prestadora de serviços hospitalares. Durante o atendimento do chamado (supervisionado por Lucas), o usuário questiona sobre o software de edição de arquivos usado na empresa, por travar muito ao formatar e salvar o arquivo. Para ambientar e auxiliar o estagiário, Lucas assumiu o atendimento, fechou o programa e o executou novamente, porém o software continuou travando. Ele então acionou o Gerenciador de Tarefas e identificou que o software de edição de arquivo ainda estava em execução. O estagiário fez os seguintes questionamentos: o que aconteceu para que o programa não tenha sido finalizado corretamente? Como fazer para conseguir executar o software corretamente? No caso de o software travar durante o salvamento e formatação, a implementação de threads auxiliaria o processo? Não se esqueça de compilar todas as informações obtidas para lhe auxiliar no desenvolvimento de um algoritmo de escalonamento de processos por meio de semáforos.

Para que você consiga responder esse e outros questionamentos, nesta seção vamos conhecer mais sobre os processos e os conteúdos pertinentes a este tema.

Bons estudos!

Não pode faltar

Um dos conceitos principais em sistemas operacionais gira em torno de processos.

Um processo pode ser definido como um programa em execução, porém o seu conceito vai além desta definição (MACHADO; MAIA, 2007).

Nos computadores atuais, o processador funciona como uma linha de produção executando vários programas ao mesmo tempo de forma sequencial, como ler um livro on-line, baixar um arquivo e navegar na internet. A CPU é responsável por alternar os programas, executando-os por dezenas ou centenas de milissegundos, para que cada um tenha acesso ao processamento, dando a ilusão ao usuário de paralelismo ou pseudoparalelismo (TANENBAUM, 2003). O pseudoparalelismo é a falsa impressão de que todos os programas estão executando ao mesmo tempo, mas na verdade o que acontece é que um processo em execução é suspenso temporariamente para dar lugar ao processamento de outro processo e assim sucessivamente.

Segundo Tanenbaum (2003), para tratar o paralelismo de forma mais fácil, foi desenvolvido um modelo responsável por organizar os programas executáveis em processos sequenciais. Um processo pode ser definido como um programa em execução incluindo os valores do contador de programa atual, registradores e variáveis. A CPU alterna de um processo para outro a cada momento, essa alternância é conhecida como multiprogramação.

A diferença entre processos e programa é importante para que o modelo seja entendido. A seguinte analogia é utilizada para exemplificar a diferença entre processo e programa: fazer um bolo. Para fazer um bolo, é necessário todos os ingredientes e a receita. A receita pode ser considerada como o programa, os ingredientes são os dados de entrada e a pessoa que prepara o bolo é o processador. Os processos são as atividades que a pessoa faz durante a preparação do bolo: ler a receita, buscar os ingredientes, misturar a massa e colocar o bolo para assar, que é o processo final desse programa "receita de bolo". Ainda neste exemplo, imagine que o filho da pessoa que está fazendo o bolo tenha se machucado. A pessoa guarda em que parte do processamento parou e vai socorrer o filho. Ela pega o kit de primeiros socorros e lê o procedimento

para tratar do machucado do filho. Neste momento, vemos o processador (a pessoa) alternando de um processo (fazer o bolo) para outro processo com prioridade maior (socorrer o filho), cada um com seu programa – receita versus procedimento de tratamento do machucado. Assim que o filho estiver medicado, então a pessoa retornará a fazer o bolo do ponto em que parou.

Podemos considerar então que um processo é uma atividade que contém um programa, uma entrada, uma saída e um estado. Veremos a seguir a criação de processo e os estados dos processos.



Exemplificando

Os serviços que os sistemas operacionais podem implementar através de processos são (MACHADO; MAIA, 2007):

- Auditoria e segurança do sistema.
- A contabilização do uso de recursos.
- A contabilização de erros.
- Gerência de impressão.
- Comunicação de eventos.
- Serviços de redes.
- Interface de comandos (Shell), entre outros.

Criação de processos

Os sistemas operacionais devem oferecer formas para que processos sejam criados. Segundo Tanenbaum (2003), existem quatro eventos que fazem com que um processo seja criado:

1) **Início do sistema:** quando o sistema operacional é inicializado, são criados vários processos. Existem os de primeiro plano, que interagem com os usuários e suas aplicações, e os de segundo plano, que possuem uma função específica, como um processo para atualizar e-mails quando alguma mensagem é recebida na caixa de entrada.

Para visualizar os processos em execução no Windows, pressione as teclas CTRL+ALT+DEL e no Linux utilize o comando ps.

2) **Execução de uma chamada ao sistema de criação por um processo em execução:** por exemplo, quando um processo está fazendo

download, ele aciona um outro processo para ajudá-lo. Enquanto um faz o download, o outro está armazenando os dados em disco.

3) **Uma requisição do usuário para criar um novo processo:** quando o usuário digita um comando ou solicita a abertura de um ícone para a abertura de um aplicativo.

4) **Início de um *job* em lote:** esses processos são criados em computadores de grande porte, os mainframes.



Assimile

Vimos nesta seção que existem os processos de primeiro plano, que são processos dos usuários e interagem com suas aplicações, e os de segundo plano, que possuem função específica. Os processos de segundo plano, ao executarem uma função específica, são chamados de *daemons*.

Término de processos

Após a criação, os processos podem ser finalizados nas seguintes condições, (TANENBAUM, 2003):

1) **Saída normal (voluntária):** acontece quando o processo acaba de executar por ter concluído seu trabalho.

2) **Saída por erro (voluntária):** acontece quando o processo tenta acessar um arquivo que não existe e é emitida uma chamada de saída do sistema. Em alguns casos, uma caixa de diálogo é aberta perguntando ao usuário se ele quer tentar novamente.

3) **Erro fatal (involuntário):** acontece quando ocorre um erro de programa, por exemplo, a execução ilegal de uma instrução ou a divisão de um número por zero. Neste caso, existe um processo com prioridade máxima que supervisiona os demais processos e impede a continuação do processo em situação ilegal.

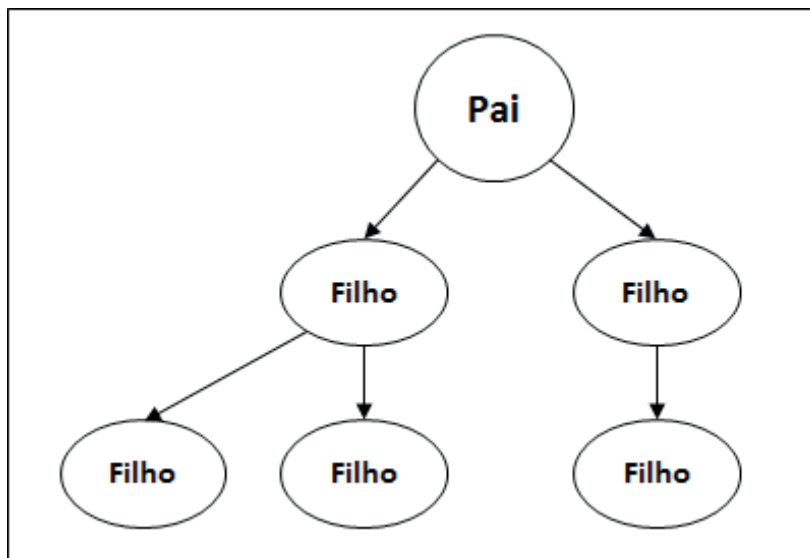
4) **Cancelamento por um outro processo:** acontece quando um processo que possui permissão emite uma chamada ao sistema para cancelar outro processo.

Hierarquia de Processos

Segundo Tanenbaum (2003), em alguns sistemas, quando um processo cria outro, o processo-pai e o processo-filho ficam

associados. O filho pode gerar outros processos, criando, assim, uma hierarquia de processos, conforme apresentado na Figura 2.1.

Figura 2.1 | Hierarquia de processos



Fonte: elaborada pelo autor.

No Unix, um processo-pai, seus filhos e descendentes formam um grupo de processos. Por exemplo, quando um usuário envia um sinal do teclado (como *CTRL + ALT + DEL*), este sinal é entregue para todos os processos que compõem o grupo de processos do teclado. Quando um processo-pai é "morto", todos os filhos vinculados a ele são "mortos" também.

O Windows não possui uma hierarquia de processos. Cada um possui um identificador próprio e quando um processo cria outro, existe uma ligação entre eles, mas ela é quebrada quando o processo-pai passa seu identificador para outro processo. Quando um processo-pai é "morto", os processos vinculados a ele não são mortos.

Estados do Processo

Os processos podem passar por diferentes estados ao longo do processamento. Um processo ativo pode estar em três estados (MACHADO; MAIA, 2007):

- **Em execução:** um processo está em execução quando está sendo processado pela CPU. Os processos são alternados para a utilização do processador.
- **Pronto:** um processo está no estado de pronto quando possui todas as condições necessárias para executar e está aguardando. O sistema operacional é quem define a ordem e os critérios para execução dos processos.
- **Espera ou Bloqueado:** um processo está no estado de espera quando aguarda por um evento externo (um comando do usuário, por exemplo) ou por um recurso (uma informação de um dispositivo de entrada/saída, por exemplo) para executar.

Quatro mudanças podem acontecer entre os estados, representadas na Figura 2.2.

Figura 2.2 | Transições dos processos



Fonte: Tanenbaum (2003, p. 58).

Conforme apresentado na Figura 2.2, a mudança 1 ("Em execução" para "Bloqueado") acontece quando um processo aguarda um evento externo ou uma operação de entrada/saída e não consegue continuar o processamento.

As mudanças 2 ("Em execução" para "Pronto") e 3 ("Pronto" para "Em Execução") são realizadas pelo escalonador sem que o processo saiba. O escalonador de processos é o responsável por decidir em qual momento cada processo será executado (veremos as atividades do escalonador com detalhes na seção 2.3). A mudança 2 acontece quando o escalonador decide que o processo já teve tempo suficiente em execução e escolhe outro processo para executar. A mudança 3 ocorre quando os demais processos já

utilizaram o seu tempo de CPU e o escalonador permite a execução do processo que estava aguardando.

A mudança 4 (“Bloqueado” para “Pronto”) ocorre quando a operação de entrada/saída ou o evento externo que o processo estava esperando ocorre. Assim, o processo retorna para a fila de processamento e aguarda novamente a sua vez de executar.

Threads

Segundo Machado e Maia (2007), o conceito de thread foi introduzido para reduzir o tempo gasto na criação, eliminação e troca de contexto de processos nas aplicações concorrentes, assim economizando recursos do sistema como um todo.

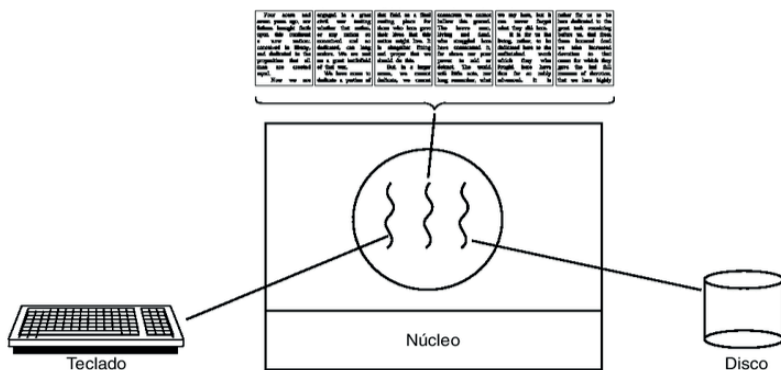
Thread é um fluxo de controle (execução) dentro do processo, chamado também de processos leves. Um processo pode conter um ou vários threads que compartilham os recursos do processo.

A principal razão para o uso de thread é que as aplicações da atualidade rodam muitas atividades ao mesmo tempo e quando são compostas por threads, podem ser executadas em paralelo. Outro motivo para a criação de thread é que são mais fáceis de criar e destruir, por não terem recursos vinculados a eles. Em relação ao desempenho, quando uma aplicação processa muitas informações de entrada/saída, o uso de threads acelera a execução da aplicação.

Para entender melhor o conceito de thread, vamos apresentá-lo utilizando como exemplo um processador de texto (TANENBAUM, 2003).

Imagine que você esteja escrevendo um livro e que seja necessário trocar um termo errado escrito em diversos capítulos. Se o livro estiver em um único arquivo, fica fácil fazer a alteração, porém se estiver em arquivos separados, será necessário fazer a correção em cada um. Outra questão é: se uma ou duas páginas forem removidas de um arquivo de 500 páginas, o processador de textos terá que formatar todo o documento em todas as páginas até chegar à página retirada, o que causará uma grande demora. Neste caso, se forem utilizados threads, um thread auxiliaria a formatação do texto, outro atenderia aos requisitos do usuário via teclado e um terceiro thread faria o backup dos dados em disco sem interferir nas ações dos demais threads, conforme apresentado na Figura 2.3.

Figura 2.3 | Processador de texto com três threads



Fonte: Tanenbaum (2003, p. 64).

Outro exemplo do uso de threads é em um navegador web, enquanto um thread carrega imagens ou textos de uma página e outro thread recupera dados de uma rede.



Pesquise mais

Algumas linguagens de programação da atualidade, como Java, possuem recursos para a implantação de threads, chamados programação concorrente. Para saber mais sobre esse assunto, leia o artigo: LANHELLAS, R. **Trabalhando com Threads em Java**. DEVMEDIA, [s.l.], 2013. Disponível em: <<https://www.devmedia.com.br/trabalhando-com-threads-em-java/28780>>. Acesso em: 11 jul. 2018.

Implementação de Processos

Para implementar o modelo de processos, o sistema operacional mantém um quadro de processos contendo informações sobre o estado do processo, seu contador de programa, o ponteiro da pilha, a alocação de memória, o status dos arquivos abertos, entre outros, que permitem que o processo reinicie do ponto em que parou (TANENBAUM, 2003). O Quadro 2.1 apresenta os campos mais importantes. No primeiro campo estão os dados necessários para o armazenamento, referentes ao gerenciamento de processos. O segundo e o terceiro campos referem-se aos dados do gerenciamento de memória e ao gerenciamento de arquivos.

Gerenciamento de processos	Gerenciamento de memória	Gerenciamento de arquivos
Registradores	Ponteiro para o segmento de código	Diretório-raiz
Contador do programa	Ponteiro para o segmento de dados	Diretório de trabalho
Palavra de estado do programa	Ponteiro para o segmento de pilha	Descritores de arquivos
Ponteiro de pilha		Identificador (ID) do usuário
Estado do processo		Identificador (ID) do grupo
Prioridade		
Parâmetros de escalonamento		
Registrador Identificador (ID) de processadores		
Processo-pai		
Grupo do processo		
Sinais		
Momento em que o processo iniciou		
Tempo usado da CPU		
Tempo de CPU do filho		
Momento do próximo alarme		

Fonte: Tanenbaum (2003, p. 59).

Implementação de Threads

A implementação de threads pode ocorrer no espaço do usuário, no núcleo do sistema operacional e em uma implementação híbrida (no espaço do usuário e do núcleo).

- **Thread de usuário:** são implementados pela aplicação do usuário e o sistema operacional não sabe de sua existência. A vantagem é que não é necessária nenhuma mudança entre os modos de usuário e núcleo, tornando-se rápido e eficiente.

- **Thread do núcleo:** são implementados e gerenciados pelo núcleo do sistema operacional. A desvantagem desta implementação é que todo o gerenciamento dos threads é feito por chamadas ao sistema, o que compromete a performance do sistema.
- **Threads híbridos:** são implementados tanto no espaço do usuário, quanto no núcleo do sistema operacional. O sistema operacional sabe dos threads do usuário e faz o seu gerenciamento. A vantagem desta implementação é a flexibilidade em função das duas implementações.



Reflita

Diante do que aprendemos sobre processos e threads, quais são os desafios de implementar processos e threads em sistemas operacionais multiprogramados?

Sem medo de errar

Agora que você já aprendeu sobre os conceitos, as características e o funcionamento de processo e threads, sobre como os processos são criados e finalizados pelo sistema operacional, sobre a hierarquia e os estados de processos e sobre como os processos e threads são implementados, vamos voltar ao nosso contexto? Lucas estava auxiliando o atendimento realizado por um estagiário de sua equipe e surgiram os seguintes questionamentos: o que aconteceu para que o programa não tenha finalizado corretamente? Como fazer para conseguir executar o software corretamente? No caso de o software travar durante o salvamento e a formatação, a implementação de threads auxiliaria o processo? O problema de travamento de um software pode acontecer tanto em função do próprio software, quanto por causa de um hardware. Como Lucas identificou que o processo estava em execução ao acionar o Gerenciador de Tarefas, porém estava travado, constatou-se o problema no processo que estava rodando. Neste caso, o processo travou por algum erro encontrado, por exemplo, na memória durante a gravação do arquivo feita pelo editor de texto. O correto é “matar” o processo

e reiniciar o software de edição. Ao fazer isso, verifique se outro processo em execução impactará no salvamento do arquivo e, em caso positivo, feche-o antes de reiniciar o software de edição. A implementação de threads durante a execução do processo agiliza o processamento e melhora o desempenho das aplicações. O próprio conceito de threads diz “são as entidades programadas para a execução sobre a CPU” (TANENBAUM, WOODHULL, 2008, p.78). Logo, ao adicionar threads em uma aplicação, esta conseguirá processar duas ou mais coisas ao mesmo tempo. Por exemplo, no caso do software de edição de arquivos, um thread seria responsável por receber os comandos do usuário via teclado, outro thread seria responsável por formatar o arquivo e outro seria responsável por salvar o arquivo, todos os threads trabalhando em sincronia e executando as atividades em paralelo.

Avançando na prática

Erro fatal ao abrir um software

Descrição da situação-problema


Um outro chamado foi aberto para tratar o erro do Adobe Reader, pois o programa travou e fechou inesperadamente. Lucas, durante o atendimento, reiniciou o computador para ver se resolveria o problema, porém não resolveu. Assim, o usuário questionou: por que esse erro aconteceu e por que a reinicialização do computador não resolveu o problema, mesmo tendo “matado” o processo?

Resolução da situação-problema

O travamento e o fechamento inesperado de softwares podem acontecer por vários motivos, como conflito de hardware e software e dados corrompidos em arquivos. Nem sempre reiniciar o computador resolverá o problema. No caso relatado pelo usuário em relação ao travamento e ao fechamento inesperado do Adobe Reader, alguns processos de segundo plano, quando executados juntamente com ele, podem causar erros. É necessário fechar esses processos de segundo plano e reiniciar o computador para que o Adobe Reader seja executado. Para isso, vá ao Gerenciador de

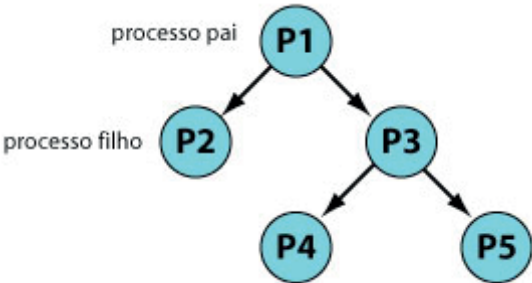
Tarefas, clique na aba Processos, selecione o processo e clique em Finalizar processo. Após esta ação, reinicie o computador e abra novamente o Adobe Reader.

Faça valer a pena

1.
- 

No Unix, processos podem gerar outros processos, dando a nomenclatura de processo “pai” e processo “filho” gerando assim uma hierarquia, alertando que processos filhos possuem apenas um processo pai, mas um processo pai pode ter vários processos filhos. O fato de ser um processo filho não impede que o mesmo também tenha processos filhos, veja na figura um exemplo de hierarquia onde o processo P1 está no topo: (SANCHES, 2012)

Figura 2.4 | Hierarquia de processos no UNIX



Fonte: Sanches (2012).

SANCHES, R. O. Hierarquia de Processos no Unix e Windows. DEVMEDIA, [s.l.], 2012. Disponível em: <<https://www.devmedia.com.br/hierarquia-de-processos-no-unix-e-windows/24739>>. Acesso em: 11 jul. 2018.

Analise as asserções a seguir:

A hierarquia de processos do Unix dificulta a propagação de vírus nos sistemas operacionais.

Porque

Quando um processo-pai é “morto” seja pelo sistema operacional ou pelo próprio usuário, todos os processos que estiverem abaixo dele na hierarquia serão “mortos” também.

Com relação às duas asserções assinale a alternativa correta:

- a) As asserções I e II são proposições verdadeiras, e a II é uma justificativa da I.
- b) As asserções I e II são proposições verdadeiras, mas a II não é uma justificativa da I.
- c) A asserção I é uma proposição verdadeira, e a II é uma proposição falsa.
- d) A asserção I é uma proposição falsa, e a II é uma proposição verdadeira.
- e) As asserções I e II são proposições falsas.

2.

Os sistemas operacionais executam de maneiras diferentes os processos e threads. No caso do Windows, ele trabalha com maior facilidade para gerenciar programas com apenas um processo e diversos threads do que quando gerencia vários processos e poucos threads. Isso acontece porque no sistema da Microsoft a demora para criar um processo e alterná-los é muito grande. Enquanto isso, o Linux e os demais sistemas baseados no Unix podem criar novos processos de maneira muito mais rápida. No entanto, ao serem alterados, os programas podem apresentar o mesmo desempenho tanto no Linux quanto no Windows.



CANALTECH. O que é Thread? [S.l.], [s.d.]. Disponível em: <<https://canaltech.com.br/produtos/o-que-e-thread/>>. Acesso em: 11 jul. 2018.

Considerando as características entre processos e threads, escolha a opção correta:

- a) Durante a execução de um processo, caso ocorra algum erro e o processo seja finalizado, a thread continua executando, independentemente do processo.
- b) Threads podem ser escalonadas pelo sistema operacional e rodar como entidade independente dentro do processo.
- c) A facilidade que o Linux tem em criar processos é devido ao processo ser criado fora do kernel, o que melhora o desempenho do sistema operacional.
- d) A dificuldade do Windows de criar processos e alterná-los é em função do grande volume de vírus que impedem que o desempenho do sistema operacional seja satisfatório.
- e) Um processo pode conter vários threads e cada thread tem autonomia dentro do processo para realizar suas atividades, sendo que os demais threads não veem as mudanças realizadas por ele.

3.



Um processo passa por diferentes estados desde sua criação até seu término. Enquanto ele é criado, seu estado é considerado "Novo"; em ação, muda para "Executando"; quando depende da ocorrência de algum evento, vira "Esperando"; quando não mais necessário, o processo é "Terminado".

AMOROSO, D. O que são processos de um sistema operacional e por que é importante saber. TecMundo, [s.l.], 4 dez. 2009. Disponível em: <<https://www.tecmundo.com.br/memoria/3197-o-que-sao-processos-de-um-sistema-operacional-e-por-que-e-importante-saber.htm>>. Acesso em: 11 jul. 2018

Sobre os estados dos processos, marque a alternativa correta.

- a) Um processo no estado pronto significa que está executando (utilizando recursos da CPU).
- b) Um processo ativo pode estar em somente dois estados diferentes: pronto ou em execução.
- c) Um processo é considerado bloqueado (esperando) quando ele não pode prosseguir em função da alocação de outro processo ocupando a CPU pelo sistema operacional.
- d) Um processo não pode passar do estado pronto para o estado em execução, pois esta transição não é permitida.
- e) Um processo em execução não significa que ele está rodando e utilizando a CPU. O processo pode estar com o estado em execução e aguardando uma entrada externa, por exemplo um comando do usuário.

Seção 2.2

Comunicação entre processos

Diálogo aberto

Caro aluno, em um sistema operacional, os processos e threads trocam informações entre si ou solicitam a utilização de recursos simultaneamente, como arquivos, dispositivos de entrada/saída e memória. Um exemplo de comunicação interprocessos é a transferência de dados entre processos. Se um processo deseja imprimir um arquivo, ele o insere em um diretório de impressão com um nome para identificá-lo e outro processo é responsável por verificar periodicamente se existem arquivos a serem impressos. Nessa seção você saberá como é feita a comunicação entre processos e threads. Veremos alguns pontos sobre essa comunicação, como condições de disputa, regiões críticas e exclusão mútua com espera ociosa. Além disso, estudaremos os mecanismos de sincronização que resolvem a exclusão mútua: dormir e acordar, semáforos, monitores e troca de mensagens. Relembrando nosso contexto, Lucas acabou de ser contratado como gerente da área de Tecnologia da Informação de uma empresa prestadora de serviços hospitalares e, durante o acompanhamento do atendimento realizado por um estagiário, em relação aos constantes travamentos do software de edição de arquivos ao salvar ou realizar a formatação, eles observam que outro usuário está fazendo o download de um arquivo. Durante o download, um processo (A) é responsável por baixar o arquivo e outro processo (B) é responsável por gravar estas informações em disco. Caso tenha outro processo acessando o disco, o processo (B) aguarda a sua vez para poder acessar o disco e gravar as informações. Caso contrário, o processo (B) grava os dados em disco e libera o recurso compartilhado para outro processo. Dessa forma, o estagiário faz os seguintes questionamentos a Lucas: quais são os desafios da comunicação entre os processos? Quais os problemas clássicos da comunicação entre processos? E se fosse utilizado um algoritmo de escalonamento de processos por meio de semáforos? Qual a importância e os benefícios de sua utilização? Para que

você consiga responder esse e outros questionamentos sobre a comunicação entre processos e os mecanismos de sincronização que resolvem a exclusão mútua, nesta seção vamos conhecer mais sobre os conteúdos pertinentes a este tema.

Bons estudos!

Não pode faltar

Em uma aplicação concorrente (execução cooperativa de processos e threads), os processos precisam se comunicar entre eles, então solicitam o uso de recursos como memória, arquivos, dispositivos de entrada/saída e registros. Por exemplo, uma região de memória é compartilhada entre vários processos. O sistema operacional deve garantir que esta comunicação seja sincronizada para manter o bom funcionamento do sistema e a execução correta das aplicações.

Segundo Tanenbaum (2003), é necessário levar em consideração três tópicos:

1. Como um processo passa a informação para outro processo.
2. Garantir que dois ou mais processos não invadam uns aos outros quando estão em regiões críticas (será detalhada no decorrer da seção). Quando um processo estiver usando uma região de memória, o outro processo deve aguardar a sua vez.
3. É necessário existir uma hierarquia quando houver dependências. Se o processo A produz dados e o processo B os imprime, B deve esperar até que A produza dados para serem impressos.

Estas questões serão discutidas ao longo desta seção e também se aplicam a threads.

Condições de disputa ou condições de corrida

Condições de disputa ou condições de corrida acontecem quando dois ou mais processos estão compartilhando alguma região da memória (lendo ou escrevendo dados) e o resultado final depende das informações de quem executa e quando.

Como exemplo, podemos citar o problema da Conta_Corrente relatado por Machado e Maia (2007). Nesta situação, o saldo

bancário de um cliente é atualizado por meio de um programa após o lançamento de um débito ou crédito no arquivo de contas correntes (neste arquivo são armazenadas informações sobre o saldo dos correntistas do banco). O registro do cliente e o valor depositado ou sacado são lidos por um programa e o saldo do cliente é atualizado. Suponha que dois funcionários do banco atualizem o saldo do mesmo cliente simultaneamente. O processo do primeiro funcionário lê o registro do cliente e soma ao saldo o valor sacado pelo cliente. Porém, antes de gravar o novo saldo no arquivo, o segundo funcionário lê o registro do mesmo cliente que está sendo atualizado e lança um crédito a ser somado ao saldo. Independentemente do processo que atualizar primeiro, o dado gravado no arquivo referente ao saldo está inconsistente.

Regiões críticas

Para impedir as condições de disputa, é necessário definir maneiras que impeçam que mais de um processo leia e escreva ao mesmo tempo na memória compartilhada. Esses métodos são chamados de exclusão mútua, ou seja, quando um processo estiver lendo ou gravando dados, sua região crítica ou processo deve esperar.

A parte do programa em que o processo acessa a memória compartilhada é chamada de região crítica ou seção crítica.

Segundo Tanenbaum (2003), para termos uma boa solução, é necessário satisfazer quatro itens:

1. Dois ou mais processos jamais estarão ao mesmo tempo em suas regiões críticas.
2. Não se pode afirmar nada sobre o número e a velocidade de CPUs.
3. Nenhum processo que esteja executando fora de sua região crítica pode bloquear outros processos.
4. Nenhum processo deve esperar sem ter uma previsão para entrar em sua região crítica.

A seguir veremos as soluções propostas para realizar a exclusão mútua: exclusão mútua com espera ociosa, dormir e acordar, semáforos, monitores e troca de mensagens.

Exclusão mútua com espera ociosa

Segundo Tanenbaum (2003), existem alguns métodos que impedem que um processo invada outro quando um deles está em sua região crítica. São eles:

Desabilitando interrupções

Nesta solução, as interrupções são desabilitadas por cada processo (qualquer parada que pode ocorrer por um evento) assim que entra em sua região crítica e ativadas novamente antes de sair dela. Desta forma, a CPU não será disponibilizada para outro processo.

Esta solução não é prudente, uma vez que, ao dar autonomia para processos, a multiprogramação fica comprometida. Se um processo, ao entrar em sua região crítica, desabilitasse as interrupções e se esquecesse de habilitá-las novamente ao sair, o sistema estaria comprometido. Em sistemas com múltiplos processadores, a interrupção acontece em apenas um processador e os outros acessariam normalmente a memória compartilhada, comprometendo essa solução.



Assimile

A solução de exclusão mútua com espera ociosa, como desabilitando interrupções e instrução TSL são mecanismos implementados no hardware do computador. As demais soluções são implementadas via software.



Exemplificando

Suponha que um processo de atualização da base de dados de uma empresa de vendas de seguros de vida desabilite as interrupções durante a noite para fazer uma nova carga da base de dados. Esta interrupção é necessária, uma vez que a carga demorará aproximadamente dez horas. Se o processo finalizar e não reabilitar as interrupções, a base de dados no dia seguinte não poderá ser usada e a consequência pode ser a perda financeira e de fidelização de clientes.

Variáveis de impedimento

Esta solução contém uma variável chamada *lock*, inicializada com o valor 0. Segundo Tanenbaum (2003), o processo testa e verifica o valor dessa variável antes de entrar na região crítica e, caso o valor seja 0, o processo a altera para 1 e entra na região crítica. Caso o valor da variável seja 1, o processo deve aguardar até que seja alterado para 0.

Variáveis de impedimento não resolvem o problema de exclusão mútua e ainda mantêm a condição de disputa. Quando o processo 1 vê o valor da variável 0 e vai para alterar o valor para entrar na região crítica, chega o processo 2 e altera o valor da variável para 1, antes de o processo 1 ter alterado. Logo, os dois processos entram, ao mesmo tempo, na região crítica.

Alternância obrigatória

Segundo Tanenbaum (2003), essa solução utiliza uma variável *turn* compartilhada que informa qual processo poderá entrar na região crítica (ordem). Esta variável deve ser alterada para o processo seguinte, antes de deixar a região crítica. Suponha que dois processos desejam entrar em sua região crítica. O processo A verifica a variável *turn* que contém o valor 0 e entra em sua região crítica. O processo B também encontra a variável *turn* com o valor 0 e fica testando continuamente para verificar quando ela terá o valor 1.

O teste contínuo é chamado de espera ociosa, ou seja, quando um processo deseja entrar em sua região crítica, ele examina se sua entrada é permitida e, caso não seja, o processo fica esperando até que consiga entrar. Isso ocasiona um grande consumo de CPU, podendo impactar na performance do sistema.

Ainda segundo Tanenbaum (2003), assim que o processo A deixa sua região crítica, a variável *turn* é atualizada para 1 e permite que o processo B entre em sua região crítica. Vamos supor que o Processo B é mais ágil e deixa a região crítica. Os processos A e B estão fora da região crítica e *turn* possui o valor 0. O processo A finaliza antes de ser executado em sua região não crítica. Como o valor de *turn* é 0, o processo A entra de novo na região crítica, e o processo B ainda permanece na região não crítica. Ao deixar a

região crítica, o processo A atualiza a variável *turn* com o valor 1 e entra em sua região não crítica.

Os processos A e B estão executando na região não crítica e o valor da variável *turn* é 1. Se o processo A tentar entrar de novo na região crítica, não conseguirá, pois o valor de *turn* é 1. Desta forma, o processo A fica impedido pelo processo B, que **NÃO** está na sua região crítica. Esta situação viola a seguinte condição: nenhum processo que esteja executando fora de sua região crítica pode bloquear outros processos.

Solução de Peterson

Segundo Tanenbaum (2003), essa solução foi implementada por meio de um algoritmo que consiste em dois procedimentos escritos em C, baseado na definição de duas primitivas (*enter_region* e *leave_region*) utilizadas pelos processos que desejam utilizar sua região crítica. Antes de entrar na região crítica, todo processo chama *enter_region* com os valores 0 ou 1. Este apontamento faz com que o processo aguarde até que seja seguro entrar. Depois de finalizar a utilização da região crítica, o processo chama *leave_region* e permite que outro entre. Como a solução de Alternância Obrigatória, a Solução de Peterson precisa da espera ociosa.

Instrução TSL

Segundo Tanenbaum (2003), a instrução TSL (*test and set lock*, ou seja, teste e atualize a variável de impedimento) conta com a ajuda do hardware.

A instrução TSL RX, LOCK faz uma cópia do valor do registrador RX para LOCK. Um processo somente pode entrar em sua região crítica se o valor de LOCK for 0. A verificação do valor de LOCK e sua alteração para 0 são realizadas por instruções ordinárias.

A solução de Alternância Obrigatória, a Solução de Peterson e a instrução TSL utilizam a espera ociosa.

Dormir e acordar

As soluções apresentadas até aqui utilizam a espera ociosa (os processos ficam em um laço ocioso até que possam entrar na

região crítica). Para resolver este problema, são realizadas chamadas *sleep* (dormir) e *wakeup* (acordar) ao sistema, que bloqueiam/desbloqueiam o processo, ao invés de gastar tempo de CPU com a espera ociosa.

A chamada *sleep* faz com que o processo que a chamou durma até que outro processo o desperte, e a chamada *wakeup* acorda um processo.

Para exemplificar o uso dessas chamadas, vamos apresentar o problema do produtor/consumidor.

Problema do produtor/consumidor

De acordo com Tanenbaum (2003), o problema do produtor/consumidor normalmente acontece em programas concorrentes em que um processo gera informações (produtor) para uso de outro processo (consumidor).

Imagine que dois processos compartilham um *buffer* (memória) de tamanho fixo. O problema acontece quando o produtor quer inserir um novo item, porém o *buffer* está cheio ou o consumidor deseja remover um item e o *buffer* está vazio.

A solução é colocar o processo, impedido pela capacidade do *buffer*, para dormir (através da chamada *sleep*) até que o outro modifique o *buffer* e acorde o anterior (por meio da chamada *wakeup*). Para controlar a quantidade de itens no *buffer*, é utilizada uma variável *count*.

O problema da condição de disputa acontece quando o *buffer* está vazio e o consumidor verifica o valor da variável *count*. O escalonador decide parar a execução do consumidor e executa o produtor, que inclui itens no *buffer* enviando um sinal de acordar para o consumidor. Como o consumidor ainda não está logicamente adormecido, o sinal para acordar é perdido e tanto o consumidor, quanto o produtor ficarão eternamente adormecidos.

Semáforos

Segundo Machado e Maia (2007), a utilização de semáforos é um dos mecanismos utilizados em projetos de sistemas operacionais e em aplicações concorrentes. Hoje, grande parte das linguagens

de programação disponibiliza procedimentos para que semáforos sejam utilizados.

Um semáforo é uma variável inteira que realiza duas operações: DOWN (decrementa uma unidade ao valor do semáforo) e UP (incrementa uma unidade ao valor do semáforo). As rotinas DOWN e UP são indivisíveis e executadas no processador. Um semáforo com o valor 0 indica que nenhum sinal de acordar foi salvo e um valor maior que 0 indica que um ou mais sinais de acordar estão pendentes (TANENBAUM, 2003).

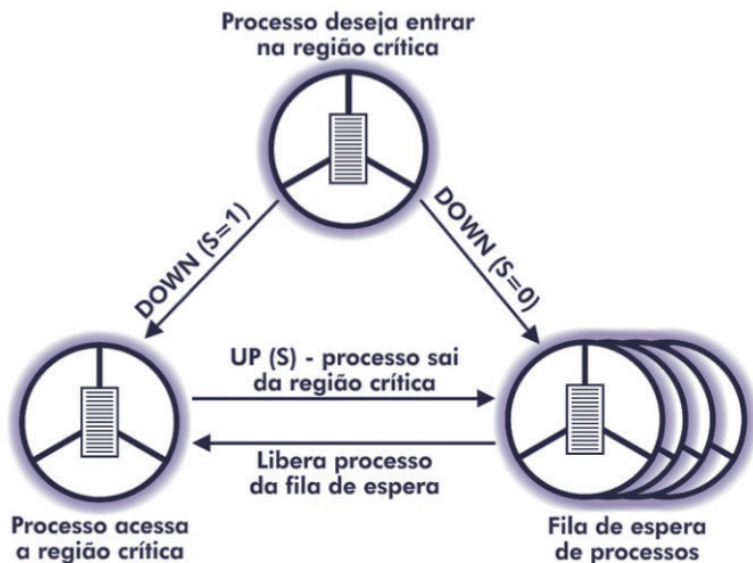
De acordo com Machado e Maia (2007), os semáforos são classificados como:

- Binários, também conhecidos como mutexes (*mutual exclusion semaphores*), que recebem os valores 0 ou 1.

- Contadores, que recebem qualquer valor inteiro positivo, além do 0.

Para compreendermos melhor o uso de semáforos, utilizaremos um exemplo de semáforos binários. Segundo Machado e Maia (2007), o semáforo com o valor igual a 1 significa que nenhum recurso está utilizando o processo e valor igual a 0 significa que o recurso está em uso. A Figura 2.5 apresenta o uso do semáforo binário na exclusão mútua. Quando um processo deseja entrar em sua região crítica, é executada a instrução DOWN. Caso o valor do semáforo seja igual a 1, o valor é decrementado e o processo pode entrar em sua região crítica. Caso o valor seja 0 e a operação DOWN seja executada, o processo é impedido de entrar em sua região crítica, permanecendo em fila no estado de espera. O processo que utiliza o recurso executa a instrução UP ao deixar a região crítica incrementa o valor do semáforo e libera o acesso ao recurso. Caso existam processos aguardando na fila para serem executados, o sistema selecionará um e alterará o seu estado para pronto.

Figura 2.5 | Semáforo binário



Fonte: Machado e Maia (2007, p. 108).

O problema do produtor/consumidor (perda de sinal de acordar) apresentado acima pode ser resolvido através de semáforos, conforme apresentado na Figura 2.6, em que três semáforos são usados: um chamado *mutex*, que controla o acesso à região crítica e é inicializado com o valor 1, outro chamado *full*, que conta os valores preenchidos no *buffer*, sendo o valor inicial 0 e o terceiro chamado de *empty*, que conta os lugares vazios no *buffer* (representa o número de lugares do *buffer*). Segundo Tanenbaum (2003), um processo, ao executar a operação **DOWN**, decrementa o seu valor inteiro. Caso o valor do semáforo seja negativo, o processo é impedido e inserido ao fim da fila. Quando um processo executa a operação **UP**, o seu valor inteiro é incrementado. Se existir algum processo impedido na fila, o primeiro processo é liberado.

Figura 2.6 | Problema do produtor/consumidor usando semáforos

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

/ número de lugares no buffer */*
/ semáforos são um tipo especial de int */*
/ controla o acesso à região crítica */*
/ conta os lugares vazios no buffer */*
/ conta os lugares preenchidos no buffer */*

/ TRUE é a constante 1 */*
/ gera algo para pôr no buffer */*
/ decresce o contador empty */*
/ entra na região crítica */*
/ põe novo item no buffer */*
/ sai da região crítica */*
/ incrementa o contador de lugares preenchidos */*

/ laço infinito */*
/ decresce o contador full */*
/ entra na região crítica */*
/ pega o item do buffer */*
/ deixa a região crítica */*
/ incrementa o contador de lugares vazios */*
/ faz algo com o item */*

Fonte: Tanenbaum (2003, p. 82).



Pesquise mais

Para saber mais sobre o funcionamento de semáforos, acesse:

ROJAS, A. **Semáforos**. 8 out. 2014. Disponível em: <<https://youtu.be/8YTV7cMyOSU>>. Acesso em: 12 jul. 2018.

Monitores

Segundo Tanenbaum (2003), um monitor é uma coleção de rotinas, estrutura de dados e variáveis que ficam juntos em um módulo ou pacote. Também pode ser definido como uma unidade de sincronização de processos de alto nível.

Um processo, ao chamar uma rotina de um monitor, verifica se existe outro processo ativo. Caso esteja, o processo que chamou é bloqueado até que o outro deixe o monitor, senão, o processo que o chamou poderá entrar.

A utilização de monitores garante a exclusão mútua, uma vez que só um processo pode estar ativo no monitor em um determinado momento e os demais processos ficam suspensos até poderem estar ativos. O compilador é o responsável por definir a exclusão mútua nas entradas do monitor.

É preciso definir métodos para suspenderem os processos caso não possam prosseguir, mesmo que a implementação da exclusão mútua em monitores seja fácil.

Assim, é necessário introduzir variáveis condicionais, com duas operações: *wait* e *signal*. Se um método do monitor verifica que não pode prosseguir, um sinal *wait* é emitido (bloqueando o processo), permitindo que outro processo que estava bloqueado acesse o monitor. A linguagem de programação Java suporta monitores.



Refleta

Os monitores e semáforos são soluções para CPUs que utilizam memória compartilhada. E como seria esse processo em sistemas distribuídos, em que existem troca de informações entre processos que estão em máquinas diferentes?

Troca de Mensagens

Segundo Tanenbaum (2003), esse método utiliza duas chamadas ao sistema:

- *send (destination, &message)* - envia uma mensagem para um determinado destino.

- *receive (source, &message)* - recebe uma mensagem de uma determinada origem.

Caso nenhuma mensagem esteja disponível, o receptor poderá ficar suspenso até chegar alguma.

A troca de mensagens possui problemas como sua perda pela rede. Para evitar este problema, assim que uma mensagem é recebida, o receptor enviará uma mensagem de confirmação de recebimento. Caso o receptor receba e não confirme o recebimento, não será problema, uma vez que as mensagens originais são numeradas de forma sequencial.

Uma questão importante refere-se à autenticação, pois é necessário saber se a fonte é real. Além disso, as mensagens enviadas e recebidas não podem ser ambíguas.

Quanto ao desempenho, copiar mensagens é um procedimento mais lento do que realizar operações sobre semáforos ou monitores. Uma solução seria realizar a troca de mensagens através de registradores.



Assimile

Vimos nesta seção que, para resolver o problema de exclusão mútua, existem as seguintes soluções: exclusão mútua com espera ociosa, dormir e acordar, semáforos, monitores e troca de mensagens. Estes métodos são equivalentes quando implementados em um único processador.



Pesquise mais

Existem diversos problemas clássicos da comunicação interprocessos, como o problema do jantar dos filósofos e o problema do barbeiro sonolento. Para saber mais sobre eles, leia: TANENBAUM, A. S. **Sistemas Operacionais Modernos**. 2. ed. São Paulo: Pearson, 2003.

Sem medo de errar

Agora que você já aprendeu como os processos e threads fazem a comunicação, conheceu as questões sobre a comunicação entre processos como condições de disputa, regiões críticas e exclusão mútua com espera ociosa e estudou sobre os mecanismos de sincronização que resolvem a exclusão mútua dormir e acordar, semáforos, monitores e troca de mensagens, vamos voltar ao nosso contexto. Durante o atendimento do chamado referente

aos constantes travamentos do software de edição de arquivos ao salvar ou realizar a formatação, eles observam que outro usuário está fazendo o download de um arquivo. Como os sistemas operacionais precisam manter a comunicação entre processos para um bom funcionamento do sistema e para não comprometer a execução das aplicações, os desafios são grandes. Os processos que concorrem por recursos do computador devem ter suas execuções sincronizadas, garantindo o correto processamento das aplicações. Para isso, o sistema operacional deve implementar mecanismos de controle e sincronismo, como a exclusão mútua que consiste em não permitir que dois processos acessem o mesmo recurso ao mesmo tempo, garantindo assim que nenhum processo invada a região crítica de outro. Neste caso, foram desenvolvidas soluções para evitar a exclusão mútua, como: **dormir e acordar**, que faz com que um processo durma enquanto outro esteja acessando o recurso, até que seja acordado pelo outro processo; **semáforo**, uma variável responsável por contar a quantidade de sinais de dormir e acordar, sendo vantajoso por não existir a espera ocupada que desperdiça o processamento de máquina; entre outras soluções. Analisando esta situação, um dos mecanismos de sincronização de processos chama-se Semáforos. Os Semáforos garantem que processos não acessem um mesmo recurso simultaneamente através de duas variáveis *Down* e *Up*, que são verificadas e alteradas sem a possibilidade de interrupções. A variável *Down* é decrementada quando o processo grava os dados e libera o disco para outro processo. A variável *Up* é incrementada quando outro processo já está usando a CPU.

Existem os problemas clássicos da comunicação entre processos e existem vários métodos para resolver estes problemas. Podemos citar o problema do jantar dos filósofos e o problema do barbeiro adormecido, conforme apresentado em Tanenbaum (2003). O problema do jantar dos filósofos foi proposto e resolvido assim: cinco filósofos estão sentados em uma mesa circular e cada um tem um prato de espaguete e garfos. Porém, para comer são necessários dois garfos, pois o espaguete está escorregadio. Cada filósofo alterna entre comer e pensar e, quando algum deles fica com fome, tenta pegar os garfos da sua esquerda e direita, um de cada vez, para poder comer o espaguete. Caso consiga pegar os dois garfos, ele come por um tempo, liberando os garfos ao final

e voltando à sua atividade de pensar. O problema é não gerar o bloqueio dos filósofos implementando soluções para resolver a questão do acesso exclusivo a um número restrito de recursos. O problema do barbeiro adormecido acontece em uma barbearia. A barbearia tem um barbeiro, uma cadeira de barbeiro e várias cadeiras para que os clientes esperem a sua vez. Se não tiver clientes, o barbeiro adormece e quando um cliente chega, o barbeiro acorda e corta seu cabelo. Se chegarem outros clientes, eles verificam se há cadeiras de espera disponíveis e, caso não tenha, os clientes vão embora. O problema principal é programar o barbeiro e os clientes para não caírem em condição de disputa.

Avançando na prática

Condições de Disputa – venda de ingressos para a Copa do Mundo 2018

Descrição da situação-problema

O estagiário que trabalha na área de Tecnologia da Informação da empresa prestadora de serviços hospitalares comentou com Lucas que um de seus colegas é o responsável pelo monitoramento do sistema de vendas de ingressos para a Copa do Mundo de 2018. O estagiário comentou sobre a falha no sistema que ocasionou a venda duplicada de ingresso, problema que ocorreu em um ponto de venda de ingressos no Rio de Janeiro e em outro em Nova York (EUA). Tanto o vendedor do Rio de Janeiro quanto o de Nova York viram que o último ingresso para a partida entre Brasil e Estados Unidos estava disponível para a venda, então, ambos venderam o ingresso aos torcedores. Diante da situação relatada pelo colega do estagiário, o SAC do sistema de vendas de ingressos foi acionado e a falha foi reportada à empresa desenvolvedora do software. O estagiário fez as seguintes perguntas para Lucas: por que ocorreu a condição de disputa, uma vez que os vendedores são de países diferentes? Como garantir que essa condição não aconteça? Apresente um relatório explicando a falha ocorrida no sistema de vendas de ingressos para a Copa do mundo 2018.

Resolução da situação-problema

Em um sistema unificado de vendas, normalmente a aplicação desenvolvida deve garantir que não ocorram erros. Mesmo estando em países diferentes, a aplicação deve ser íntegra, não permitindo que problemas como a condição de disputa aconteça. É possível que tenha ocorrido uma falta de sincronismo, ocasionando essa condição, uma vez que os pontos de venda dependem da internet. Para garantir que isso não aconteça, é necessário verificar se a solução de exclusão mútua (que garante que um processo não terá acesso a uma região crítica enquanto outro estiver utilizando essa região) foi corretamente implementada através do método escolhido pelo desenvolvedor da aplicação. Logo, é preciso ver com a empresa que desenvolveu o sistema para reportar o erro.

Faça valer a pena

1. O sistema operacional deve implementar formas de controle e sincronismo de processos que concorrem por recursos do computador, não permitindo que dois processos acessem o mesmo recurso ao mesmo tempo, garantindo assim a integridade do sistema e das aplicações.

Considerando as questões acerca da comunicação entre processos, marque a alternativa correta:

- a) A condição de disputa ocorre quando dois ou mais processos não podem acessar os recursos compartilhados e, com isso, há uma disputa entre os processos para utilizar o recurso.
- b) Seção crítica ou região crítica é a parte do programa que o processo não pode acessar por não ser compartilhada.
- c) Exclusão mútua é um método para evitar a condição de disputa, impedindo que mais de um processo acesse uma determinada área de memória compartilhada ao mesmo tempo.
- d) Em alguns casos, o sistema operacional permite que dois processos acessem simultaneamente sua região crítica.
- e) Se um processo estiver executando fora de sua região crítica, ele pode bloquear outro processo até finalizar a execução.

2. Existem diversas soluções que resolvem o problema da exclusão mútua durante a comunicação entre processos. Entre elas podemos citar as

soluções de hardware, em que é implementada a solução diretamente no hardware (por exemplo, desativando interrupções) e de software, que são implementadas através de algoritmos.

Em relação às soluções de exclusão mútua, marque a alternativa correta.

- a) A solução de desativar as interrupções é uma das mais eficazes, pois nenhuma interrupção de relógio pode ocorrer e o processo pode finalizar a sua execução.
- b) A solução de Peterson é implementada através de uma variável e utilizada para permitir a entrada de um processo na região crítica quando um outro já está na região.
- c) A solução de dormir e acordar faz com que um processo “durma” até ser “acordado” por outro processo. Um dos problemas que pode ser resolvido com esta solução é a do produtor/consumidor.
- d) Quando um processo deseja entrar na região crítica, é atribuído um valor para a variável *lock*, em que o 0 significa que existem processos na região crítica e 1 significa que não existem processos. Essa solução é conhecida como semáforo.
- e) Na solução de monitores, vários processos podem estar ativos dentro do monitor em um mesmo instante.

3.



Há ocasiões onde é extremamente necessário que dois ou mais processos/threads acessem um único recurso comum. Caso esse tipo de paralelismo não ocorra de forma controlada, podemos fazer com que um processo “sequestre/atropele” a operação de outro. É aí que entram nossos sinalizadores, vulgo semáforos. Com eles, é possível acesso controlado a processos, de forma que só haverá disponibilidade quando a operação em andamento for finalizada.

PEREIRA, J. W. Semáforos, Kernel, Memória Compartilhada e Cia. Dicas-L, [s.l.], 21 set. 2005. Disponível em: <http://www.dicas-l.com.br/arquivo/semaforos_kernel_memoria_compartilhada_e_cia.php#.WtTCJH9ryUk>. Acesso em: 12 jul. 2018.

Considerando as questões acerca dos semáforos, marque a alternativa correta.

- a) O semáforo é um mecanismo que resolve problemas apenas de sincronismo e não de exclusão mútua em sistemas operacionais.

- b) Um semáforo não pode ser usado em linguagens orientadas a objetos por ser de difícil implementação.
- c) Quando um processo muda o valor de um semáforo, um outro processo pode alterar este valor simultaneamente.
- d) Uma vez que uma operação sobre semáforos começa, ela não pode ser interrompida no meio e nenhuma outra operação sobre o semáforo deve ser iniciada.
- e) A variável Down decrementa o valor do semáforo e não é necessário testar se é menor que 0, visto que a implantação de semáforos é muito segura.

Seção 2.3

Escalonamento de processos

Diálogo aberto

Prezado aluno, em um sistema operacional vários processos compartilham recursos ao mesmo tempo, e quem faz a escolha de qual processo deve ser executado é o escalonador, feita por meio de um algoritmo (algoritmo de escalonamento), sendo necessário seguir as seguintes premissas: dar a cada processo o tempo necessário de uso da CPU, verificar se a política estabelecida é cumprida e manter ocupadas todas as partes do sistema. Por exemplo, quando um processo termina sua execução, outro processo deve ser escolhido entre os que estão no estado "pronto para executar". Nessa seção você verá como é realizado o escalonamento entre processos e threads e estudará os tipos de escalonamento e seus principais algoritmos. Relembrando nosso contexto, Lucas acabou de ser contratado como gerente da área de Tecnologia da Informação de uma empresa prestadora de serviços hospitalares. Um usuário abriu um chamado para atualização do Adobe Reader em seu computador. Durante o atendimento, Lucas acompanha a resposta do chamado, delegada a um estagiário, que inicia a atualização do software e observa que no computador do usuário existem dois processos ativos: A e B. O processo A vai atualizar o Adobe Reader, enquanto o processo B vai gravar em disco os dados da alteração de um arquivo iniciada pelo usuário. Neste caso, o escalonador, além de fazer todas as verificações necessárias, deve conferir a prioridade de execução, pois a gravação em disco é prioritária em relação à atualização de um aplicativo. O escalonador de processos deve escolher um algoritmo de escalonamento para realizar esta ação, e Lucas foi designado para implementar um algoritmo de escalonamento de processos por meio de semáforos, que deverá ser entregue ao diretor da área de TI do hospital. Dessa forma, o estagiário fez os seguintes questionamentos: quais critérios o escalonador leva em consideração na escolha do uso da CPU? Quais são os objetivos do algoritmo de escalonamento? Por que

em alguns algoritmos de escalonamento é necessário saber o tempo de execução de cada processo para executar? Como os sistemas operacionais atuais fazem a escolha do algoritmo de escalonamento a ser utilizado? A fim de conseguir responder essas e outras questões sobre o escalonamento de processos e sobre os algoritmos de escalonamento, vamos conhecer mais sobre eles, para que você consiga implementar um algoritmo de escalonamento de processos através de semáforos.

Bons estudos!

Não pode faltar

Segundo Tanenbaum (2003), nos computadores existem vários processos que competem pela CPU e é necessário que o sistema operacional escolha, de forma eficiente, os que estejam aptos a executar. O responsável por isso é o escalonador de processos, por meio da aplicação de algoritmos ou políticas de escalonamento para otimizar a utilização do processador, definindo o processo que ocupará a CPU.

Segundo Machado e Maia (2007), além de escolher o processo a ser executado, o escalonador deve prezar pelos critérios e pelos objetivos (Quadro 2.2).

Quadro 2.2 | Critérios e objetivos do escalonador de processos

Critérios	<ul style="list-style-type: none">- Utilização do processador: eficiência do uso da CPU mantendo o processador ocupado na maior parte do tempo.- Throughput: maximizar a produtividade (<i>throughput</i>), executando o maior número de processos em função do tempo.- Tempo de processador: tempo de execução do processo.- Tempo de espera: reduzir o tempo total que um processo aguarda na fila para ser executado.- Tempo de turnaround: minimizar o tempo que um processo leva desde sua criação até seu término, considerando a alocação de memória, tempo de espera e tempo do processador e aguardando as operações de entrada/saída.- Tempo de resposta: reduzir o tempo de resposta para as aplicações interativas dos usuários.
------------------	---

Objetivos

- Dar privilégios para aplicações críticas.
- Balancear o uso da CPU entre processos.
- Ser justo com todos os processos, pois todos devem poder usar o processador.
- Maximizar a produtividade (*throughput*).
- Proporcionar menores tempos de resposta para usuários interativos.

Fonte: adaptado de Machado e Maia (2007).

Diferentes sistemas operacionais apresentam características de escalonamento distintas. Podemos citar como exemplos o sistema operacional em tempo real e o de tempo compartilhado. O primeiro prioriza as aplicações críticas, enquanto o segundo aloca todos os processos com tempo igual para acesso à CPU, a fim de que os processos não esperem muito tempo para ter acesso ao processamento.

Segundo Tanenbaum (2003), alternar processos é oneroso, uma vez que é necessário alternar do modo usuário para o modo núcleo para iniciar a execução. Nessa execução, o estado do processo e o mapa de memória devem ser salvos, armazenando os dados dos registradores na tabela de processos e, a cada troca de processos, a memória cache (memória de acesso rápido) é invalidada.

As principais situações que levam ao escalonamento, segundo Tanenbaum (2003), são:

- **A criação de um novo processo:** é necessário escolher entre executar o processo pai ou o filho.
- **O término de um processo:** quando um processo é finalizado, é necessário escolher outro para ser executado.
- **Bloqueio do processo:** quando um processo é bloqueado e está aguardando uma entrada/saída, é necessário escolher outro processo.
- **Interrupção de entrada/saída:** se a interrupção for gerada por um dispositivo que finalizou a execução, o processo passará de "bloqueado" para "pronto" e o escalonador deve escolher entre continuar executando o processo atual ou o que acabou de ficar pronto.
- **Interrupções de relógio:** a cada interrupção do hardware de relógio pode haver um escalonamento de processos.

Em relação ao tratamento das interrupções de relógio, os algoritmos ou políticas de escalonamento são classificados em não-preemptivo e preemptivo (MACHADO; MAIA, 2007). No não-preemptivo um processo executa até finalizar, independentemente do tempo de uso da CPU, ou até que seja bloqueado aguardando entrada/saída de outro processo. Este escalonamento foi implementado no processamento batch. Já no escalonamento preemptivo um processo é executado por um tempo pré-determinado e quando o tempo de execução dado ao processo finaliza, a CPU é alocada para outro processo. No escalonamento preemptivo é possível priorizar aplicações em tempo real em função dos tempos dados aos processos. Os algoritmos de escalonamento preemptivo são complexos, porém permitem a implantação de vários critérios de escalonamento.



Refleta

Já pensou na complexidade e na criticidade de se definir a quantidade certa de tempo de execução de cada processo (*quantum*)? Por que esse é um processo crítico? Quais seriam as implicações se este processo fosse malfeito?

Segundo Tanenbaum (2003), existem três ambientes diferentes de escalonamento: lote, interativo e tempo real. A seguir descreveremos cada um deles.

- Lote: como não existem usuários aguardando uma resposta, tanto algoritmos preemptivos como não-preemptivos são aceitáveis para sistemas em lote. Os algoritmos de escalonamento aplicados aos sistemas em lote são:

- **FIFO** (*First in first out*): primeiro a chegar, primeiro a sair. Neste algoritmo os processos são inseridos em uma fila à medida que são criados, e o primeiro a chegar é o primeiro a ser executado. Quando um processo bloqueia e volta ao estado de pronto, ele é colocado no final da fila e o próximo processo da fila é executado. O FIFO é um algoritmo não-preemptivo, simples e de fácil implementação.
- **Job mais curto primeiro** (SJF – *shortest job first*): é um algoritmo de escalonamento não-preemptivo, em que

são conhecidos todos os tempos de execução dos *jobs*. O algoritmo seleciona primeiro os *jobs* mais curtos para serem executados. Este algoritmo é recomendado quando todos os *jobs* estão disponíveis ao mesmo tempo na fila de execução. A Figura 2.6 (a) apresenta quatro *jobs* (A, B, C e D) aguardando numa fila, com os respectivos tempos de execução em minutos (8, 4, 4 e 4). Se os *jobs* forem executados nesta ordem, teremos uma média de espera de execução de 14 minutos (o retorno do *job* A é de 8 minutos, o retorno do *job* B é de 12 minutos (8 + 4), o retorno do *job* C é de 16 minutos (12 + 4) e o retorno do *job* D é de 20 minutos (16 + 4). Logo, $(8+12+16+20) / 4 = 14$ minutos. Se os *jobs* forem executados selecionando primeiramente o mais curto, conforme apresentado na Figura 2.6 (b), teremos uma média de espera de execução de 11 minutos (o retorno do *job* B é de 4 minutos, o do *job* C é de 8 minutos (4 + 4), o do *job* D é de 12 minutos (8 + 4) e o do *job* A é de 20 minutos (12 + 8)). Logo, $(4+8+12+20) / 4 = 11$ minutos.

Figura 2.6 | Escalonamento do *job* mais curto primeiro



Fonte: Tanenbaum (2003, p. 102).

Uma versão preemptiva para o algoritmo *job* mais curto é, primeiro, o algoritmo próximo de menor tempo restante. O escalonador conhece os tempos de execução e escolhe sempre o *job* cujo tempo restante ao seu término seja o menor. Quando um novo *job* chega na fila para execução, seu tempo total é comparado ao tempo restante do processo que está utilizando a CPU.

- **Interativo:** nos sistemas interativos, a preempção se faz necessária para que outros processos tenham acesso à CPU. Os algoritmos de escalonamento aplicados aos sistemas interativos e que podem também ser aplicados a sistema em lote são:

- Escalonamento Round Robin: é um algoritmo antigo, simples, justo e muito usado. Também é conhecido como algoritmo de escalonamento circular. Nele os processos são organizados em uma fila e cada um recebe um intervalo de tempo máximo (*quantum*) que pode executar. Se ao final de seu *quantum* o processo ainda estiver executando, a CPU é liberada para outro processo. A Figura 2.7 (a) mostra a lista de processos que são executáveis mantida pelo escalonador. Quando um processo finaliza o seu *quantum*, é colocado no final da fila, conforme apresentado na Figura 2.7 (b).

Figura 2.7 | Escalonamento Round Robin



Fonte: Tanenbaum (2003, p. 104).

Segundo Machado e Maia (2007), o *quantum* varia de acordo com a arquitetura do sistema operacional e a escolha deste valor é fundamental, uma vez que afeta a política do escalonamento circular. Os valores variam entre 10 e 100 milissegundos. O escalonamento circular é vantajoso porque não permite que um processo monopolize a CPU.

- **Escalonamento por prioridades:** o algoritmo por prioridades considera todos os processos importantes, sendo associados a ele uma prioridade e um tempo máximo de execução. Por exemplo, um processo que carrega os dados em uma página web deve ter uma prioridade maior do que um processo que atualiza em segundo plano as mensagens de correio eletrônico. Quando os processos estiverem disponíveis para execução, o que tiver a maior prioridade é selecionado para executar. Para que os processos com prioridades altas não sejam executados infinitamente, a cada interrupção de relógio o escalonador pode reduzir a prioridade do processo.

Segundo Machado e Maia (2007), as prioridades de execução podem ser classificadas em estática ou dinâmicas.

A prioridade estática não altera o valor enquanto o processo existir. Já a dinâmica ajusta-se de acordo com os critérios do sistema operacional.

- **Escalonamento garantido:** se existirem vários usuários (n) logados em uma máquina, cada um deles receberá $1/n$ do tempo total da CPU. O sistema gerencia a quantidade de tempo de CPU de cada processo desde sua criação.



Exemplificando

Para exemplificar o escalonamento garantido, imagine que em um computador multiusuário existam 4 usuários logados. O tempo de CPU estabelecido para cada usuário executar os seus processos são de 7 segundos. Assim, o processo do usuário 1 executa por 7 segundos e para, dando lugar ao 2, que também executa por 7 segundos. Inicia então o processo do usuário 3 que executa pelo mesmo tempo e para, iniciando o processo do usuário 4. Este ciclo se repete até que todos os processos dos usuários finalizem a sua execução.

- **Escalonamento por loteria:** o escalonamento por loteria é baseado em distribuir bilhetes aos processos e os prêmios recebidos por eles são recursos de sistema, incluindo tempo de CPU. Cada bilhete pode representar o direito a um *quantum* de CPU e cada processo pode receber diferentes números de bilhetes, com opções de escolha distintas. Também existem as ações como compra, venda, empréstimo e troca de bilhetes.
- **Escalonamento fração justa (*fair-share*):** nesse caso, cada usuário recebe uma fração da CPU. Por exemplo, se existem dois usuários conectados em uma máquina e um deles tiver nove processos e o outro tiver apenas um, não é justo que o usuário com o maior número de processos ganhe 90% do tempo da CPU. Logo, o escalonador é o responsável por escolher os processos que garantam a fração justa.



O LinSched é um simulador de *scheduler* do Linux que reside no espaço do usuário. Ele isola o subsistema do *scheduler* do Linux (que fica dentro do kernel) e desenvolve ao redor dele uma quantidade suficiente do ambiente do kernel para que possa ser executado dentro do espaço do usuário. Para saber mais sobre este *scheduler*, acesse: JONES, M. **Simulação do Linux Scheduler**. IBM, 12 abr. 2011. Disponível em: <<https://www.ibm.com/developerworks/br/library/l-linux-scheduler-simulator/index.html>>. Acesso em: 13 jul. 2018.

- **Tempo real:** nesses sistemas, o tempo é um fator importantíssimo e os processos, ao utilizarem a CPU, fazem seu trabalho rapidamente e são bloqueados, dando oportunidade para outros processos executarem. Segundo Machado e Maia (2007), o escalonamento por prioridades seria o mais adequado em sistemas de tempo real, uma vez que uma prioridade é vinculada ao processo e, assim, a importância das tarefas na aplicação são consideradas. No escalonamento de tempo real, a prioridade deve ser estática, além de não existir fatia de tempo para cada processo executar.



Vimos nesta seção que no escalonamento de tempo real, o tempo é um fator crucial. Esse sistema pode ser classificado em crítico e não crítico. O crítico atende a todos os prazos de um processo. No não crítico, os prazos de tempo do processo não podem ser garantidos.

Escalonamento de Threads

Da mesma forma que processos são escalonados, threads também são. O escalonamento de threads depende se estas estão no espaço do usuário ou do núcleo. Se forem threads de usuário, o núcleo não sabe de sua existência e o sistema operacional escolhe um processo A para executar, dando a ele o controle de seu *quantum*. O escalonador do thread A escolhe qual thread deve executar, através dos algoritmos de escalonamento descritos anteriormente. Se forem threads do núcleo, o sistema

operacional escolhe um thread para executar até um *quantum* máximo e, caso o *quantum* seja excedido, o thread será suspenso (TANENBAUM, 2003).

Uma das diferenças entre threads do usuário e do núcleo é o desempenho, uma vez que a alternância entre threads do usuário e do núcleo consome poucas instruções do computador. Além disso, os threads do usuário podem utilizar um escalonador específico para uma aplicação (TANENBAUM, 2003).

Segundo Deitel, Deitel e Choffnes (2005), na implementação de threads em Java, cada thread recebe uma prioridade. O escalonador em Java garante que o thread com prioridade maior execute o tempo todo. Caso exista mais de um thread com prioridade alta, eles serão executados através de alternância circular.

Sem medo de errar

Agora que você já aprendeu sobre como é realizado o escalonamento de processos e threads e como os algoritmos de escalonamento funcionam, vamos relembrar o nosso contexto.

Durante a atualização do Adobe Reader no computador do usuário, enquanto o processo A atualiza o software, o processo B grava em disco os dados da alteração de um arquivo iniciada pelo usuário. O escalonador de processos deve escolher um algoritmo para realizar esta ação, e Lucas foi designado para implementar um algoritmo de escalonamento de processos por meio de semáforos, que deverá ser entregue ao diretor da área de TI da empresa prestadora de serviços hospitalares. Desta forma, Lucas foi questionado pelo estagiário: quais critérios o escalonador leva em consideração na escolha do uso da CPU? Quais são os objetivos do algoritmo de escalonamento? Por que em alguns algoritmos de escalonamento é necessário saber o tempo de execução de cada processo para executar? Como os sistemas operacionais atuais fazem a escolha do algoritmo de escalonamento a ser utilizado?

Em um computador multiprogramado, ou seja, que executa vários programas ao mesmo tempo, os processos disputam a CPU, sendo necessário que o escalonador defina qual processo executará. Escolher corretamente o processo é importante, uma

vez que alternar processos é caro, sendo preciso se preocupar com o uso eficiente da CPU. Para isso, é necessário que o escalonador siga os seguintes critérios:

- Manter o processador ocupado a maior parte do tempo, prezando pela eficiência da CPU.
- Executar o maior número de processos em função do tempo.
- Reduzir o tempo total que um processo aguarda na fila para ser executado.
- Minimizar o tempo que um processo leva desde a sua criação até o seu término.
- Considerar a alocação de memória, o tempo de espera e do processador e aguardar as operações de entrada/saída.
- Reduzir o tempo de resposta para as aplicações interativas dos usuários.

Os objetivos do escalonador de processos são: dar privilégios para aplicações críticas, balancear o uso da CPU entre processos, ser justo com todos os processos, pois todos devem ter poder usar o processador, maximizar a produtividade (*throughput*) e proporcionar menores tempos de resposta para usuários interativos. Em alguns algoritmos de escalonamento, por exemplo, no **Job mais curto primeiro**, o tempo de execução de cada processo é conhecido, porque estes algoritmos eram utilizados em sistemas em lote e, nestes sistemas, era possível prever o tempo de execução de cada programa, por serem rotinas que executavam constantemente. Assim, o processo que possuía o menor tempo de execução era selecionado para ocupar a CPU.

Os sistemas operacionais atuais precisam levar em consideração o tempo de resposta rápido das aplicações dos usuários, manter em conformidade os processos com prioridades altas e baixas, entre outros. Em tempo de execução, o escalonador pode definir qual processo executará de acordo com a sua política de escalonamento. Além disso, ele deve manter a CPU ocupada na maior parte do tempo, executando o maior número de processos possíveis. Os escalonadores dos sistemas operacionais atuais são preemptivos, isto é, dividem o tempo do processador em partes e cada parte é alocada aos processos. Assim, todos os processos que chegam para executar possuem seu tempo de CPU garantido.

Após analisar o cenário da empresa prestadora de serviços hospitalares, Lucas implementou o algoritmo de escalonamento de processos por meio de semáforos Binários. Foi definido que quando nenhum recurso estiver sendo utilizado pelo processo, o valor do semáforo será igual a 1, caso o valor seja igual a 0 significa que o recurso está em uso. O Quadro 2.3 apresenta o pseudocódigo utilizado por Lucas para implementar as funções *down* e *up*. Quando um processo deseja entrar em sua região crítica, é executada a instrução *down*. Se o valor do semáforo for igual a 1, o valor é decrementado e o processo pode entrar em sua região crítica. Caso o valor seja 0 e a operação *down* seja executada, o processo é impedido de entrar em sua região crítica, permanecendo em fila no estado de espera. O processo que está utilizando o recurso executa a instrução *up* ao sair da região crítica, incrementando o valor do semáforo e liberando o acesso ao recurso. Caso existam processos aguardando na fila para serem executados, o sistema selecionará um e alterará o seu estado para pronto.

Quadro 2.3 | Pseudocódigo Semáforo

Função Down Se (semáforo = 1) então semáforo = semáforo – 1 entrar na região crítica Senão Processo bloqueado fimSe	Função UP Se (semáforo = 0) então semáforo = semáforo + 1 sair da região crítica Senão processo pronto fimSe
---	---

Fonte: elaborado pelo autor.

Avançando na prática

Escalonamento por prioridades

Descrição da situação-problema

A empresa C&C, prestadora de serviços de TI como suporte via Service Desk e desenvolvimento de websites para pequenas e médias empresas de todos os ramos, possui filiais em Brasília, São

Paulo, Rio de Janeiro e Belo Horizonte. Diariamente, ocorrem videoconferências para alinhamento de procedimentos técnicos e para tratar de assuntos específicos dos projetos de TI das empresas para as quais a C&C presta serviços. As videoconferências são realizadas por um notebook que fica na sala de reuniões. João é o técnico de TI responsável pela videoconferência da sede da C&C e observou que o tempo de resposta da videoconferência da sua filial estava muito lento. João entrou no gerenciador de tarefas do computador e identificou que o processo responsável pela videoconferência não estava executando com a prioridade correta. Observou também que o processo de atualização de e-mails tinha uma prioridade superior ao da videoconferência. Diante do exposto, surgem os seguintes questionamentos: como um processo responsável pela videoconferência pode ter prioridade inferior ao processo de atualização de e-mails? A alteração de prioridade foi realizada pelo sistema operacional ou por outra pessoa? Como fazer para alterar a prioridade dos processos em execução?

Resolução da situação-problema

Um sistema de videoconferência trabalha em tempo real, sendo o tempo um fator importantíssimo. Em um computador, as prioridades são definidas pelo sistema operacional ou podem ser mudadas pelos usuários. Como relatado por João, as videoconferências são realizadas por um notebook que fica na sala de reuniões. Logo, alguém por interesse próprio alterou a prioridade da videoconferência e priorizou o processo de atualização de e-mails. Mesmo que a videoconferência tenha prioridade sobre o processo de atualização de e-mails, se o usuário fizer a alteração de prioridade, o sistema operacional vai obedecer a ordem de prioridades definida. Para alterar a prioridade dos processos, basta seguir os seguintes passos:

- No Windows: abra o Gerenciador de Tarefas e selecione o processo do qual deseja alterar a prioridade. Clique com o botão direito do mouse, selecione a opção "Definir prioridade" e clique na prioridade desejada.

- No Linux: as prioridades do Linux variam de -19 a 20. Para alterar a prioridade, basta digitar o seguinte comando: `renice -n prioridade -p $(pidof) ->` -n representa o parâmetro do comando, prioridade representa um número de -19 a 20 e o *pidof* representa o PID do processo.

Faça valer a pena

1. Os algoritmos de escalonamentos podem ser classificados em preemptivos e não-preemptivos. De acordo com esta classificação, analise as afirmações a seguir.

I – o algoritmo não-preemptivo escolhe um processo e o deixa executar até que seja bloqueado ou deixe a CPU.

II – o algoritmo preemptivo escolhe um processo e o deixa executar até um tempo pré-determinado.

III – caso não exista relógio, o algoritmo preemptivo é a única alternativa a ser escolhida.

IV – o algoritmo não-preemptivo é de fácil implementação, enquanto o preemptivo possui uma implementação mais complexa.

Assinale a alternativa correta:

- a) I, II e III estão corretas.
- b) I, III e IV estão corretas.
- c) II, III e IV estão corretas.
- d) I e II estão corretas.
- e) III e IV estão corretas.

2.



Um Escalonador de Processos é um subsistema do Sistema Operacional responsável por decidir o momento em que cada processo obterá a CPU. É utilizado algoritmos de escalonamento que estabelecem a lógica de tal decisão. Nesse momento de decidir qual escalonador será utilizado no sistema operacional, cabe avaliar o cenário que o sistema será utilizado.

NOVATO, D. **Sistemas Operacionais - O que é escalonamento de processos?** Oficina da Net, 22 maio 2014. Disponível em: <<https://www.oficinadanet.com.br/post/12781-sistemas-operacionais-o-que-e-escalonamento-de-processos>>. Acesso em: 16 jul. 2018.

Analise as asserções:

Durante a escolha do escalonador sobre qual processo executar, é necessário ter cuidado com algumas variáveis (como tempo de uso da

CPU de cada processo, redução do tempo de resposta, todos os processos devem utilizar a CPU, entre outros).

PORQUE

Os processos são únicos e imprevisíveis, sendo necessário equilibrar todas estas variáveis.

Com relação às asserções acima, assinale a alternativa correta.

- a) As asserções I e II são proposições verdadeiras, e a II é uma justificativa da I.
- b) As asserções I e II são proposições verdadeiras, mas a II não é uma justificativa da I.
- c) A asserção I é uma proposição verdadeira, e a II é uma proposição falsa.
- d) A asserção I é uma proposição falsa, e a II é uma proposição verdadeira.
- e) As asserções I e II são proposições falsas.

3. Com relação aos algoritmos de escalonamento de processos, analise as questões a seguir e associe as colunas de acordo com a característica de cada algoritmo:

- | | |
|--|---|
| 1 - FIFO | () a cada processo é atribuído um intervalo de tempo no qual ele é permitido executar. |
| 2 - <i>Job</i> mais curto primeiro | () cada processo possui uma prioridade e o processo com maior prioridade é executado primeiro. |
| 3 - Escalonamento por alternância circular | () quando um processo termina o seu intervalo de tempo, ele é colocado no final da fila de execução. |
| 4 - Escalonamento por prioridades | () o tempo de execução de todos os processos são previamente conhecidos. |
| | () o primeiro processo que solicita a CPU é o primeiro a ser alocado. |

Assinale a alternativa que possui a ordem correta da associação das duas colunas:

- a) 3, 4, 3, 1, 2.
- b) 2, 1, 3, 4, 3.
- c) 4, 3, 2, 3, 1.
- d) 1, 3, 4, 2, 4.
- e) 3, 4, 3, 2, 1.

Referências

AMOROSO, D. **O que são processos de um sistema operacional e por que é importante saber**. TecMundo, [s.l.], 4 dez. 2009. Disponível em: <<https://www.tecmundo.com.br/memoria/3197-o-que-sao-processos-de-um-sistema-operacional-e-por-que-e-importante-saber.htm>>. Acesso em: 11 jul. 2018.

CANALTECH. **O que é Thread?** Disponível em: <<https://canaltech.com.br/produtos/o-que-e-thread/>>. Acesso em: 10 abr. 2018.

DEITEL, H. M.; DEITEL, P. J.; CHOFFNES, D. R. **Sistemas Operacionais**. 3. ed. São Paulo: Prentice Hall, 2005.

JONES, M. **Simulação do Linux Scheduler**. IBM, 12 abr. 2011. Disponível em: <<https://www.ibm.com/developerworks/br/library/l-linux-scheduler-simulator/index.html>>. Acesso em: 13 jul. 2018.

LANHELLAS, R. **Trabalhando com Threads em Java**. DEVMEDIA, [s.l.], 2013. Disponível em: <<https://www.devmedia.com.br/trabalhando-com-threads-em-java/28780>>. Acesso em: 11 jul. 2018.

MACHADO, F. B.; MAIA, L. P. **Arquitetura de Sistemas Operacionais**. 4. ed. Rio de Janeiro: LTC, 2007.

NOVATO, D. **Sistemas Operacionais - O que é escalonamento de processos?** Oficina da Net, 22 maio 2014. Disponível em: <<https://www.oficinadanet.com.br/post/12781-sistemas-operacionais-o-que-e-escalonamento-de-processos>>. Acesso em: 16 jul. 2018.

PATIL, S. S. **Limitations and capabilities of Dijkstra's semaphore primitives for coordination among processes**. USA: MIT, 1971.

PEREIRA, J. W. **Semáforos, Kernel, Memória Compartilhada e Cia**. Dicas-L, [s.l.], 21 set. 2005. Disponível em: <http://www.dicas-l.com.br/arquivo/semaforos_kernel_memoria_compartilhada_e_cia.php#.WtTCJH9ryUk>. Acesso em: 12 jul. 2018.

ROCHA, R. B. Desenvolvendo aplicações concorrentes estáveis e escaláveis. **Revista Modelo Java Magazine**, [s.l.], 2009. Disponível em: <<http://summa.com.br/wp-content/uploads/2009/06/programacaoconcorrente.pdf>>. Acesso em: 12 jul. 2018.

ROJAS, A. **Semáforos**. 8 out. 2014. Disponível em: <<https://youtu.be/8YTV7cMyOSU>>. Acesso em: 12 jul. 2018.

SANCHES, R. O. **Hierarquia de Processos no Unix e Windows**. DEVMEDIA, [s.l.], 2012. Disponível em: <<https://www.devmedia.com.br/hierarquia-de-processos-no-unix-e-windows/24739>>. Acesso em: 11 jul. 2018.

TANENBAUM, A. S. **Sistemas Operacionais Modernos**, 2. ed. São Paulo: Pearson, 2003.

TANENBAUM, A. S.; WOODHULL, A. S. **Sistemas operacionais: projeto e implementação**. 3. ed. Porto Alegre, Bookman, 2008.